A CHECKPOINTING METHODOLOGY FOR ANDROID SMARTPHONE

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of the Ohio State University

By

Yunhao Bai

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2016

Master's Examination Committee:

Prof. Xiaorui Wang, Advisor

Prof. Feng Qin

© Copyright by

Yunhao Bai

2016

ABSTRACT

As today's smartphone hardware becomes increasingly powerful with more CPU cores, larger size of RAM, and even graphics multi-processors, the complexity of smartphone apps is approaching that of a desktop computer. However, Random smartphone rebooting and unexpected shutdown (due to reasons like overheating, battery run outs, or automatic app upgrades) can be frustrating. What can be even worse is that a phone user commonly has to restart, from beginning, the apps he or she is using when the phone gets rebooted, as all the app states would be lost, especially when the number of apps is large. Hence, a recovery service is sorely needed for today's smartphones where apps are becoming increasingly complex. While checkpointing has long been used for desktop and laptop computers, such state preserving techniques cannot be directly applied to smartphones, due to their more stringent constraints on energy, running time, and storage space. In this paper, we propose *CKPTriod*, an checkpointing framework that checkpoints the apps based on the apps' characteristics and future usage, in order to significantly reduce the amount of work that a user needs to recover the app states after restarting the smartphone. *CKPTriod* is prototyped on the Android platform and evaluated using real-world traces.

This document is dedicated to my dear families and friends.

ACKNOWLEDGMENTS

Without the help of the following people, I would not have been able to complete my thesis. My heartfelt thanks to:

Dr. Xiaorui Wang, for being my mentor in research. Under his guidance, I learned more professional knowledge and research skills, involved with the exiting research topics, but more importantly, I have got the chance to improved myself with a better understanding of the research methodology, ideas development and research attitude. I feel very grateful for the opportunity to have him as my research advisor.

Dr. Feng Qin, for his valuable suggestions on the algorithm model design and experiment implementation, as well as his great advices and helps on the revision of my thesis writing. I do appreciate him for all above helps.

Li Li, for the discussion and test in the optimization algorithm design.

Dr. Mai Zheng, for his unselfish suggestions and great helps in sharing his knowledge and experience of OS concept and framework implementation.

VITA

2014 . . . . B.S. Electronic Information Engineering,
Shandong University
Jinan, China

2014-Present . . . . M.S. Student, Electronic Information Engineering,
The Ohio State University
Columbus, Ohio, USA

FIELDS OF STUDY

Major Field: Electrical and Computer Engineering

TABLE OF CONTENTS

Abstract ii

Dedication iii

Acknowledgments iv

Vita v

List of Figures vii

List of Tables viii

CHAPTER PAGE

1 Introduction 1

2 Related Work 4

3 Design and Implementation 5

3.1 Design Overview 5
3.2 Event Detector 6
3.3 CP analyzer 8
3.4 Checkpointer 11

4 Evaluation 15

4.1 Experimental Methodology 15
4.2 Checkpointing Cost of Some Apps 16
4.3 Retrieving Time vs Number of Thread of Memory Size 17
4.4 Energy Cost comsuption 18
4.5 System Overhead 19

5 Conclusion 21

Bibliography 22

LIST OF FIGURES

FIGURE PAGE

3.1 *CKPTroid* Framework which consists of four parts: 1)*Event Detector*, 2) *CP analyzer*, 3) *App Selector*, and 4) *Checkpointer*. 5

3.2 An example of *maps file*. memory address range, permissions, offset and open-file name can be directly read from each column. 9

3.3 Least-Square-Error fitting result of CP analyzer. The function used for fitting is quadratic model. 11

4.1 Checkpointing time and storage cost of some most often used apps based on users' traces. 16

4.2 Retrieving time and storage cost of some most often used apps based on users' traces 16

4.3 Retrieving time vs memory size and number of threads. 18

4.4 power consumption trace of checkpointing an typical app (500MB virtual memory, 20 thread). The average power for checkpointing is 500mW-700mW. 19

LIST OF TABLES

TABLE PAGE

3.1 Detect Different Shutdown Events 6

4.1 System Overhead 20

CHAPTER 1

INTRODUCTION

A variety of apps have been developed for smartphones, such as gaming (AngryBird), social networking (Facebook) and online shopping (Amazon). Such apps have greatly enhanced the user experiences and enable smartphones to gradually replace desktop/laptop, to some degree, in people's daily lives [1].

However, compared with desktops, smartphones are reported to have more random rebooting and unexpected shutdown (due to reasons like overheating, battery run outs, or automatic app upgrades)[2], which can have a highly negative impact on the users' experiences. What can be even worse is that a smartphone user commonly has to restart, from beginning, the apps he or she is using when the smartphone gets rebooted, as all the app states would be lost. For instance, a user may use Priceline.com to book a flight on his smartphone. He types in the destination, selects the departing and returning dates, then begins browsing the related flights. After some comparison, he settles down with a flight and begins to check out. At this moment, the smartphone reboots unexpectedly (e.g, due to overheating). After the reboot, the smartphone returns to its home page. The user has to click the Priceline icon to restart the app and finds that the states of the Priceline app (e.g., selected dates and flight) are totally lost. The user has to do the search from the beginning, which can be highly frustrating. Similar scenarios could often occur in everyone's daily life. In those situations, the user would be able to continue what he was doing

before the reboot, if the app's states could be effectively preserved. Hence, a recovery service is sorely needed for today's smartphones.

Application state preserving techniques have long been used on desktop/laptop computers. For instance, hibernation is usually leveraged to preserve states when a laptop runs out of battery. During a hibernation process, all the open documents and active applications are saved to the hard drive before the laptop is shut down. Once the laptop is powered back on, everything can be resumed for the user to continue the work. Unfortunately, such techniques have not yet been adopted for smartphones. The key reason is that smartphones were traditionally regarded as just a communication gadget (such as those feature phones in the early stage). All the apps of those feature phones, like contacts and calendar, usually are really simple with limited functionality and do not have any states that must be preserved upon a reboot. Due to their much simpler designs, traditional feature phones also do not usually have many random reboots (e.g., those caused by software). However, as today's smartphones are becoming almost as complex as laptops, in terms of both hardware and software apps, the deman for state recovery is significantly increasing.

It is not easy to apply those state preserving techniques previously designed for desktops (such as checkpointing [3]) directly to smartphones, due to their much more stringent constraints on energy, running time, and storage space. According to the data reported by Yahoo [4], users have an average of 95 apps installed on their smartphones, 35 of which are used (on average) every day. In order to switch between different apps smoothly and pick up from where they leave, users often push an app to the background and switch to another one. Thus, multiple apps can often co-exist in the memory at the same time. If we directly adopt the strategy designed for desktop, i.e., checkpointing all the active apps, it would cause a high delay overhead and consumes a large storage space, as well as a considerable amount of battery

lifetime, which can lead to negative impacts on the user experience. Thus, due to such resource constraints, only a subset of smartphone apps should be checkpointed before an undesired reboot or shutdown. Such a selective checkpointing introduces a new research challenges: How to catch the reboot signals and determine the right timing to start checkpointing? Starting too late would not allow sufficient time, while starting too early can interfere with the user's normal operation. In this paper, we propose *CKPTroid*, a light-weight selective checkpointing framework that aims to checkpoint a subset of apps on smartphone. We try to migrating checkpointing tool on Linux to Android platform, and manage to build a model to estimate the cost of checkpointing an app We implement a prototype of *CKPTroid* on the Android platform and evaluate it with real user traces. We also discuss practical implementation issues, such as the possible reasons for a smartphone to reboot or shut down unexpectedly and how *CKPTroid* can help preserve states in each case.

The rest of the thesis is organized as follows. In Section 2, We review the related work. Section 3 discusses the design and implementation of *CKPTroid*. Section 4 evaluates the performance of *CKPTroid* with the app usage and traces. Section 5 concludes the thesis.

CHAPTER 2

RELATED WORK

Due to limited resource, a lot of research has been done to improve the efficiency and user experience of smartphones from different perspectives [5, 6, 7, 8, 9, 10]. In this work, we consider the resource limit of smartphones and improve the user experience from a new perspective through checkpointing apps when the smartphones shutdown/reboot to reduce the recovery efforts. Checkpointing and restart has been leveraged and studied for different purposes on various platforms. Related research has been done to provide fault tolerance for distributed applications [11, 12, 13]. In this paper, we adopt checkpointing-restart on mobile platform and consider special scenarios on smartphones. Checkpointing-restart has also been implemented on Android platform. Hof et al. [14] enable Android applications to become multi-surface through app migration. They adopt checkpoint/restore to seamlessly migrate an app from one device to another. In contrast to the existing work, we do checkpointing on Android in a different method.

CHAPTER 3

DESIGN AND IMPLEMENTATION

How to determine the *right* timing to start checkpointing and how to do checkpoint are two critical problems for selective checkpointing on smartphone. In this section, we first present the overview of *CKPTroid*, then we discuss each component in detail to show how *CKPTroid* solves the problems step by step.

3.1 Design Overview

CKPTroid is designed to checkpoint the apps running on the Android smartphone. As shown in Figure 3.1, *CKPTroid* consists of four parts: 1) Event Detector, 2) CP analyzer, 3) App Selector, and 4) Checkpointer.

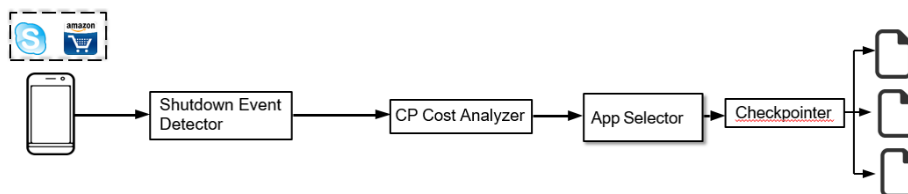

Figure 3.1: *CKPTroid* Framework which consists of four parts: 1)*Event Detector*, 2) *CP analyzer*, 3) *App Selector*, and 4) *Checkpointer*.

To better illustrate the framework, we briefly describe the workflow of *CKPTroid* as follows. Once a shutdown event is detected by Shutdown Event Detector, CP Cost Analyzer analyzes all the current running apps' checkpointing costs. Both the resource constraints and the estimated checkpointing costs are then sent to App

checkpointer. At this time, With all these inputs, App Selector selects a set of apps using greedy algorithm i.e., selects apps as many as possible. Finally, Checkpointer checkpoints the selected apps to the flash memory and recovers the states of the checkpointed apps after the smartphone reboots.

3.2 Event Detector

One critical problem is when to start checkpointing. Periodically checkpoint apps may interfere users' normal usage and the latest app states can not be saved. Thus, in this work, we perform event based checkpointing. in order to do checkpointing on triggers, we have to find out the the reasons for unexpected shutdown and the exact places that cause shutdown/reboot. Then, to do checkpointing, we insert our CKPTroid in these functions or signal catch function to perform checkpointing tasks. Some reasons of unexpected shutdown/reboot are shown in Table 3.1.

Event	How to detect
Low Battery	Instrument *shutdownIfNoPowerLocked*() in the class *BatteryService*
Overheat	Monitor the battery temperature
Watchdog	Instrument *run*() in class *Watchdog*
Native crash	Catch the *SIGSEGV* signal
Recovery mode requested	Instrument *bootCommand* in class *RecoverySystem*

Table 3.1: Detect Different Shutdown Events

Shutdown Event Detector catches the unexpected shutdown/reboot events whenever the following scenarios occur. Here we list some most often reasons for reboot or shutdown in android smartphone and the details of how to catch and detect them.

- *Low Battery or Overheating.* When the battery level is critically low or its temperature gets higher than a certain threshold, the hardware sensors of the battery will raise an alarm signal to Linux kernel, which further notifies the android framework. In real android system, when the battery runs out of power, the textitBatteryManager will notice this issue and will broadcast an intent(*INTENT_REQUEST_SHUTDOWN*) requesting the device to shut down. After that, *PowerManagerService* captures the message and performs the shutdown task. In order to do checkpointing tasks, we can insert *CKPTroid* just before the intent is broadcasted.

- *Reboot by WatchDog.* Watchdog is intended to prevent deadlock in the OS or ANR (application no response) of some most essential service to android framework like activity manager service and window manager service. For every 30 seconds, watchdog sends an monitoring heartbeat, requesting All critical respond to it If a process failed to respond for two continuous monitor request, *WatchDog* will think the OS need to be reset and kill *systemserver* process, causing Android runtime to reboot.

- *Native crash of critical processes.* Native crash happens when a fatal error, e.g., an invalid pointer or something divided by zero, is thrown by Linux kernel. If native crash happens on the processes like *zygote* or *systemserver* that are indispensable to Android systems, the OS has to reboot. For native crush, Once a SIGSEGV signal is catched by the ***Shutdown Event Detector***, the *pid* attached to SIGSEGV signal is checked: If this pid for *zygote* or *systemserver*, ***Shutdown Event Detector*** will notice this Exception and triggers checkpoint procedure.Thus, we would register a signaling handling function for SIGSEGV. In this function we'll examine the pid attached to SIGSEGV: If this pid is for an

essential process to android, then we run CKPTroid and do the checkpointing work.

- *Recovery mode is requested.* This scenario often occurs when the user installs a system update. Before rebooting the phone for the update, some preparation task must be done. After the preparation, the *PowerManagerService* class is informed and it reboots the smartphone into the recovery mode. As always, we can insert CKPTroid before the reboot action is taken in run function.

3.3 CP analyzer

CP Cost Analyzer estimates the checkpointing cost of the to-be-reused apps once a shutdown event is detected. We will discuss this component as follows.

CP Cost Analyzer calculates the checkpointing cost of an running app from the three perspectives: storage, energy and time delay. More specifically, the storage cost is for saving the app's runtime state, mainly including 1) the contents of all memory regions, 2) a list of open file descriptors and the position within the file they are pointing to, and 3) per-thread storage area descriptors and per-thread signal masks. Furthermore, the energy cost is proportional to the time needed for checkpointing an app.

The time delay cost can be divided into two parts: interruption in the app execution (the checkpointing time) and the time for the checkpoint data to asynchronously reach disk (the sync time). In turn, the checkpointing time is determined by two factors: the application's memory usage at a certain point and the number of threads an application contains [3]. The sync time only depends on the memory usage. To more accurately analyzing the checkpointing time for each app, we use a polynomial model to represent the relationship among the time delay cost, the number

of threads, and the application state size. for ***CP Cost Analyzer***, some low-level programming like querying the Linux proc filesystem or catching system error signal is required. To estimate required time for checkpoint an app, we need to know the number of threads and memory usage details of an app. To do this, We query the */proc/pid/status* file to get the number of threads in an app. Then, */proc/pid/maps* is accessed to determine the memory usage details of a process. Based on DMTCP [3], upon each line in */proc/pid/maps*, we need to determine whether this section of memory will be written in chckpoint file depending on the section's permission, anonymousity, and open-file properties. Also, we needs to peek into another process memory to find pages that are important in checkpointing an app.

```
1|root@hammerhead:/proc/180 # cat maps
40003000-40005000 r-xp 00000000 b3:19 117        /system/bin/app_process
40005000-40006000 r--p 00001000 b3:19 117        /system/bin/app_process
40006000-40007000 rw-p 00000000 00:00 0
40007000-40016000 r-xp 00000000 b3:19 177        /system/bin/linker
40016000-40017000 r--p 0000e000 b3:19 177        /system/bin/linker
40017000-40018000 rw-p 0000f000 b3:19 177        /system/bin/linker
40018000-4001b000 rw-p 00000000 00:00 0
4001b000-4001c000 r--p 00000000 00:00 0
4001c000-4001d000 rw-p 00000000 00:00 0          [anon:libc_malloc]
4001d000-40026000 r-xp 00000000 b3:19 840        /system/lib/libcutils.so
40026000-40027000 r--p 00008000 b3:19 840        /system/lib/libcutils.so
40027000-40028000 rw-p 00009000 b3:19 840        /system/lib/libcutils.so
40028000-4002b000 r-xp 00000000 b3:19 929        /system/lib/liblog.so
4002b000-4002c000 r--p 00002000 b3:19 929        /system/lib/liblog.so
4002c000-4002d000 rw-p 00003000 b3:19 929        /system/lib/liblog.so
4002d000-40074000 r-xp 00000000 b3:19 816        /system/lib/libc.so
40074000-40075000 ---p 00000000 00:00 0
40075000-40077000 r--p 00047000 b3:19 816        /system/lib/libc.so
40077000-4007a000 rw-p 00049000 b3:19 816        /system/lib/libc.so
4007a000-40088000 rw-p 00000000 00:00 0
40088000-40089000 r-xp 00000000 b3:19 1045       /system/lib/libstdc++.so
40089000-4008a000 r--p 00000000 b3:19 1045       /system/lib/libstdc++.so
4008a000-4008b000 rw-p 00001000 b3:19 1045       /system/lib/libstdc++.so
4008b000-400a3000 r-xp 00000000 b3:19 931        /system/lib/libm.so
400a3000-400a4000 r--p 00017000 b3:19 931        /system/lib/libm.so
400a4000-400a5000 rw-p 00018000 b3:19 931        /system/lib/libm.so
400a5000-400b6000 r-xp 00000000 b3:19 1057       /system/lib/libutils.so
400b6000-400b7000 ---p 00000000 00:00 0
400b7000-400b8000 r--p 00011000 b3:19 1057       /system/lib/libutils.so
400b8000-400b9000 rw-p 00012000 b3:19 1057       /system/lib/libutils.so
400b9000-400bc000 r-xp 00000000 b3:19 837        /system/lib/libcorkscrew.so
400bc000-400bd000 r--p 00002000 b3:19 837        /system/lib/libcorkscrew.so
400bd000-400be000 rw-p 00003000 b3:19 837        /system/lib/libcorkscrew.so
400be000-400c2000 r-xp 00000000 b3:19 883        /system/lib/libgccdemangle.so
400c2000-400c3000 ---p 00000000 00:00 0
400c3000-400c4000 r--p 00004000 b3:19 883        /system/lib/libgccdemangle.so
400c4000-400c5000 rw-p 00005000 b3:19 883        /system/lib/libgccdemangle.so
400c5000-400ea000 r-xp 00000000 b3:19 812        /system/lib/libbinder.so
400ea000-400f0000 r--p 00024000 b3:19 812        /system/lib/libbinder.so
400f0000-400f1000 rw-p 0002a000 b3:19 812        /system/lib/libbinder.so
400f1000-40196000 r-xp 00000000 b3:19 799        /system/lib/libandroid_runtime.
```

Figure 3.2: An example of *maps file*. memory address range, permissions, offset and open-file name can be directly read from each column.

Figure 3.2 is an example of map file. According to DMTCP checkpoint procedure, all useful information of a process must be recorded and written to the checkpoint file, this include file descriptors of a process, memory segments, shared library etc. for file descriptor information, DMTCP will call the function writefiledescrs to record all information about file descriptors that the process has open. For memory area, DMTCP determines whether a segment showed in proc/pid/maps is useful depends on property of that segment, e.g. read/write permission, anonymity and so on. The details of some skipped memory section is listed below.

- Skip anything that is in kernel space. Also, skip vdso segment because it is dangerous to restore it.

- Skip non-anonymous segmentation that do not have read/write permission, because it can be restored using file-path and offset provided in map file.

- Segments that are loaded by DMTCP are skipped because these segments are written at the beginning of the checkpoint file.

- For anonymous segments, pages that contains completely zero are skipped because they do not contain data in that process.

- Specially in android we do not need to record a non-anonymous segment if we have no write permission, because we can restore it using file-path and offset provided in map file, the exception is that the file is already deleted or region has changed.

After this procedure an typical android App that occupies 600MB virtual memory is recorded as a file of 20MB-50MB, which is crucial in restarting. For item 1 to 4, we can simply decide whether to skip it by reading the properties of each row in *map* file. However, In order to detect zero pages in the memory, we need to peek

into target process and decide zero pages in anonymous memory section. We use *P-TRACE_ATTACH* to attach to the target process. After we use *PTRACE_ATTACH*, the target process will be suspended and CP analyzer is able to access memory content of target process. For each line in map files, we assign a pointer that points the beginning of the memory section, we examine the memory content and get the memory content of the process using offset given by maps. This aims to find these pages that hold complete zero in their content in anonymous segments. Then CP analyzer detach the target process. After that, *CP Cost Analyzer* calculates the storage cost, delay cost and energy cost of checkpointing a running app.

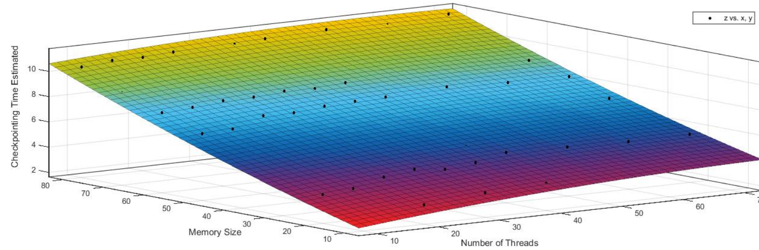

Figure 3.3: Least-Square-Error fitting result of CP analyzer. The function used for fitting is quadratic model.

In order to estimate checkpointing time, we build a static model featuring how checkpointing time is related to useful memory data and number of threads. Thus, we checkpoints some test processes and record their checkpointing time, then from these data, we build the model using least-square-error fitting. The fitting result is shown in fig 3.3.

3.4 Checkpointer

overview of DMTCP

The addition of a checkpoint/restore functionality to operation system has been an ongoing topic of discussion and development for some years now. One basic technique

is based on custom kernel modules. However, this approach limits the application that can be checkpointed in a controlled environment. Moreover, kernel modules are hard to maintain because they directly access internals of the kernel that change more rapidly than standard APIs. Thus, in recently years, a lot of userspace-based checkpoint technique have been proposed and implemented on real devices. Distributed Multi-Thread Check-Pointing (DMTCP) [3]is proposed by Jason Ansel et al at 2009 and proved to be able to work in many circumstances. Also, DMTCP runs totally in userspace, and can checkpoint multiple process simultaneously. Furthermore, in 2012 Kito modified DMTCP source code and checkpointed Zygote on android smartphone successfully [15]. Therefore, we choose DMTCP to do our simulation.

Checkpoint and Restarting Procedure

DMTCP employs an indepedent process called *dmtcp_coordinator* to manage and communicate with target processes. All target processes are connected to it and wait for its command. When a target process is started with DMTCP, DMTCP checks whether a *DMTCP_coordinator* already exists on this host: if this is true, then it will connect to the coordinator; Otherwise, DMTCP will create a new coordinator in the background. Then *LD_PRELOAD* is used to load *dmtcphijack.so* and create a *DmtcpWorker* instance before users program. The constructor of *DmtcpWorker* will place wrappers around a number of *libc* and system call implementations using function *prepareDmtcpWrappers*. Then, after some examinations and tests of the environment, a check-pointing thread will be injected into the target process by *initializeMtcpEngine* and waiting for the checkpoint request from the coordinator. Moreover, signal handling function (SIGUSR2 by default) for other user threads is registered. The users program is executed normally afterwards. Once checkpoint

request is received, all user threads are signaled by the check-pointing thread (SIGUS-R2) and begin to save their registers to the stack. Afterwards, all user thread get suspended on Futux (Linux native condition variable). Then, check-pointing thread begins to writes out all state of the process to the disk. The state consists of 1)the contents of all memory regions, including stacks, heap and any dynamically loaded libraries (via *dlopen*, etc), *mmap*ed regions, etc 2) a list of open file descriptors and the position within the file they are pointing to 3) per-thread storage area descriptors and per-thread signal masks. After finishing writing the content, all user threads are resumed and the process continues executing.

Upon a restart request, the files gets reopened, sockets gets recreated and reconnected first. Then, a new process is created by *CreateProcess*. Afterwards, memory area, socket buffer and kernel buffer gets refilled by function *mtcp_restart*. Finally, all user thread get resumed, and the process continues executing as if it had just finish writing checkpoint file.

DMTCP involves check-pointing multiple processes, thus synchronization of different processes is a big issue. In order to solve this problem, DMTCP divides checkpoint and restart progress into stages, and assigns cluster-wide barriers to each stage: After a process reaches the barrier, a *DMT_OK* message will be sent to the coordinator and the process will wait for the coordinator to release the barrier: The coordinator counts the number of *DMT_OK* message, and release the barrier if all processes *DMT_OK* message is received. This mechanism guarantee the synchronization among target processes because no process can be in next stage until all other process have reached the same barrier.

In order to transplant DMTCP to android, many features that are unique in android should be considered carefully, such as ashmem, logger, binder, pmem, etc. Also, since android uses bionic library instead of POSIX, a lot of functions are not

supported in android. Among all of these new features, binder is the one that need most supports in android. In order to support this new feature, we try to hijack the binder: When dev/binder is open, we reset the options that is set by *ioctl* before. Also, some functions should be added to DMTCP connection class such *postcheckpoint*, *precheckpoint* and *postrestart* to provide necessary hooks for DMTCP to work. As for bionic, we add some feature to stardard bionic library and uses *.patch* file to let the system build it. Thus, heavily used function in DMTCP like r/w locks and *pthread_cancel* are available in android system.

CHAPTER 4

EVALUATION

In this section, we first introduce the experimental setup and methodology, and then discuss each experiment in detail.

4.1 Experimental Methodology

We implement *CKPTroid* on a Galaxy Nexus smartphone [16] with Android version (4.4.4). A Monsoon Power Monitor [17] is used to measure the power consumption of the smartphone. All the components of *CKPTroid* are implemented as a background service. We collect the usage data from five volunteer users for two weeks, and figure out the most often used apps of users. More specifically, we design three sets of experiments as follows.

In the first set of experiments (sections 4.2), we checkpoint some apps that are most often used in users' trace and see the checkpoint time and checkpoint files' size of each app. The goal is to find the time range and average cost of checkpointing an app and determine whether complete checkpointing can be done in smartphone, Also, we measure the retrieving time of each app . In the first set of experiments, we measure the retrieving time with different number of thread and an app's memory size. We only measure retriving time because checkpointing time is measured in ***CP analyzer*** discussed in Chapter 3. Then we measure the energy cost of checkpointing

an app using power monitor. At last, we measure the the system overhead during normal use of smartphones.

4.2 Checkpointing Cost of Some Apps

In this set of experiments, we select the most often used apps that appears in the two-week usage traces from two users and do checkpointing through ADB tool. Then we retrieve them from checkpointing files.

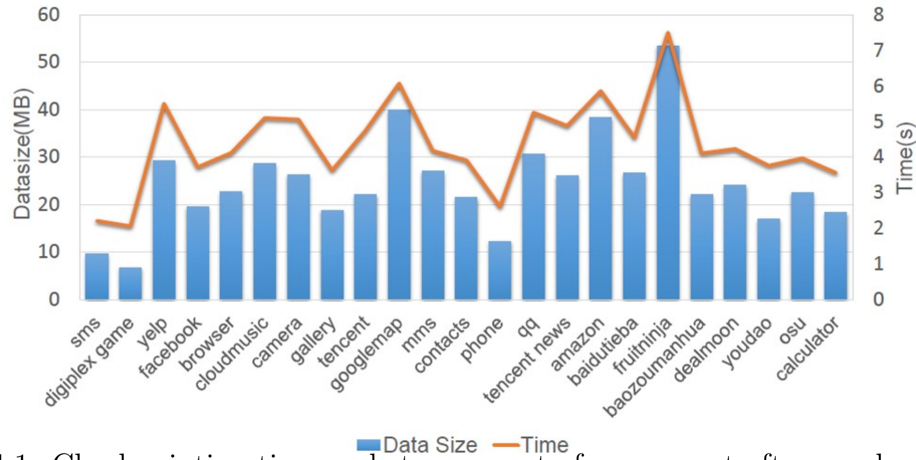

Figure 4.1: Checkpointing time and storage cost of some most often used apps based on users' traces.

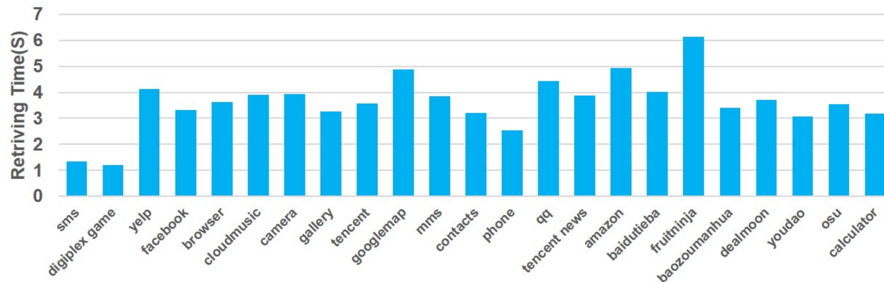

Figure 4.2: Retrieving time and storage cost of some most often used apps based on users' traces

Figure 4.1 shows the time and storage cost of checkpointing an app. The line represents checkpointing time and the bar represents the storage cost. we can see that in general, the more complex an app is, the more it will cost when it come to be checkpointed. The reason is that a complex app (games like Fruit Ninja) often has more thread and occupies more memory than a simple app(like SMS). Also, we can see that checkpointing time is nearly proportional to storage cost. It is because checkpointing time is mainly determined by sync time, i.e., the time needed for a process to write its data to the disk.

Figure 4.1 also indicted that complete checkpointing is not available on smartphone. We can see from the figure that the average checkpointing time for an app is 4 to 5 seconds. If we have 10 apps running on the smartphone on average, we'll have one minute waiting for the checkpointing task to finish: This will greatly reduce QoS of the smartphone.

Figure 4.4 shows the time needed for retrieving an app from the the checkpointing file. Retrieving time is smaller than checkpointing time because reading from the disk is much faster than writing to the disk. Moreover, the retrieving time on average is 3-4 seconds. Thus, If we do complete checkpointing, the booting process will cost 30-40 more second to finish.

4.3 Retrieving Time vs Number of Thread of Memory Size

In this section, we discuss how number of threads and memory size impacts retrieving time for an app. we write test process and put number of thread and memory size as two arguments in this program. In the program, we use *ptread_create* to create threads that loop forever in their functions; we use *malloc* to allocate data in heap and assign random number to it. DMTCP will not write heap to disk if the area is not used in the program.

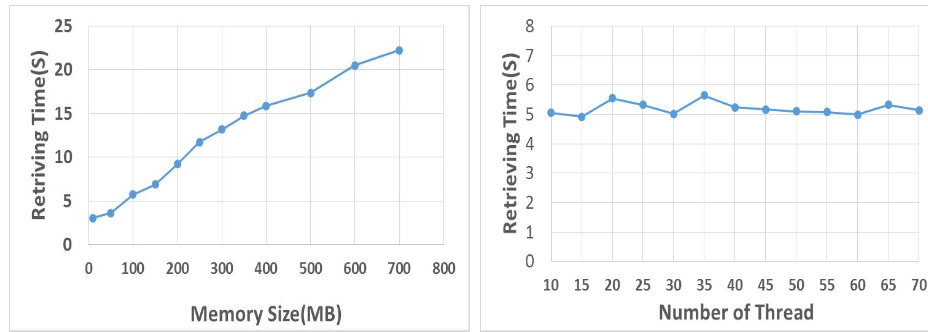

Figure 4.3: Retrieving time vs memory size and number of threads.

Figure 4.3 shows how memory size and number of thread impact retrieving time of a process on smartphone. The time for retrieving a process is greatly influenced by the memory size, but The time for retrieving does not rely on the number of threads in a process. The reason is in restarting a process, we do not need to synchronize all threads as checkpointing does: user threads are created in the last step and they can perform their tasks independently. Also, number of thread will also affect memory usage because they have their own stacks (8MB by default) stored in the memory. However, for memory restoring, all data in the checkpointing file needs to be read and put back into an process' memory content. That time dominates the retrieving time of a process which is determined by original memory size before an process get checkpointed.

4.4 Energy Cost comsuption

In this section, we measure the energy cost of checkpointing an app. This experiment aims to find whether energy consumption is a cost for checkpointing. Also, we need to make sure that *CKPTroid* could be deployed on smartphone in case of the energy cost being too big to do checkpointing.

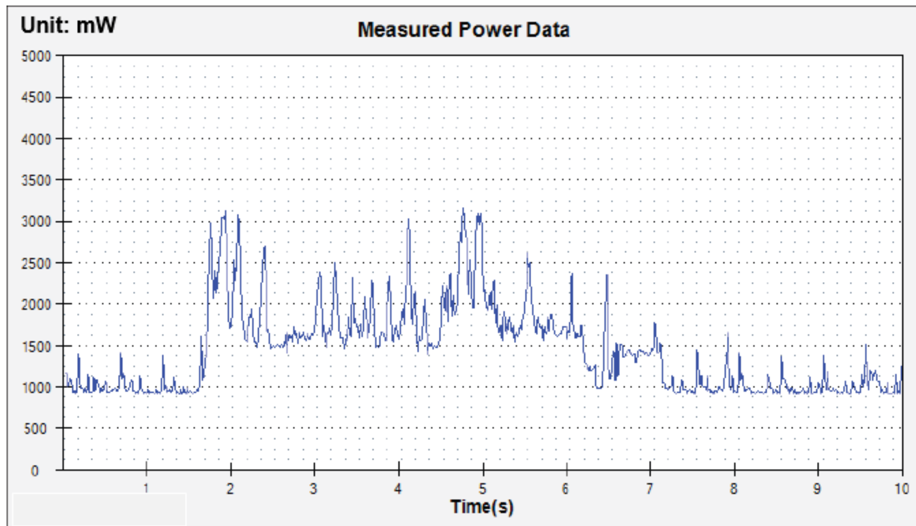

Figure 4.4: power consumption trace of checkpointing an typical app (500MB virtual memory, 20 thread). The average power for checkpointing is 500mW-700mW.

Figure 4.4 shows the energy cost trace of checkpointing an typical app(500MB memory size in total, 20 threads). we can see from the figure that the checkpoint process takes about 4 second to finish. Also, the average power of performing checkpoint is 500mW-600mW, which is very big in smartphone (GPS consumes 500mW for comparison). However, the checkpoint time is only 4-5 second, and that means the total energy consumption of checkpointing an typical app is only 2J-4J. As today's battery technic advancing, a typical smartphone battery has a capacity of 2000mAh to 3000mAh. That means checkpointing an app only consume 0.02% battery. Even if there are 30 app running, the energy checkpoint cost is still very low and can be negligible in some sence.

4.5 System Overhead

We measure the overhead of *CKPTroid* on the Galaxy Nexus Phone. Table 4.1 shows the summarized runtime overhead. The memory overhead of 13.6MB is modest and remains stable during execution. The CPU utilization is less than 1%. The power

consumption is about 3mW when the user does not interact with the smartphone, while the power consumption is 27mW when the user interacts with certain app. The time for *CKPTroid* to complete the optimization and make the decision about which app to checkpoint is less than 91ms. Overall, we can see that the overhead of *CKPTroid* is negligible to modest.

Memory	13.6MB
Processor utilization	< 1%
Power consumption (without user interaction)	3mW
Power consumption (with user interaction)	27mW
Optimization Time Cost	< 91ms

Table 4.1: System Overhead

CHAPTER 5

CONCLUSION

In this paper, we have presented *CKPTroid*, a checkpointing framework that checkpoint a subset of apps according to their cost. We find the places that cause unexpected shutdown/reboot and insert our *CKPTroid* to perform checkpointing task. We have shown that complete checkpointing is not available on smartphones. We estimate the cost of checkpointing an app and migrate DMTCP to android smartphone. We checkpoint some most often used apps on smartphone based on real users' traces and measure the energy consumption of cheeckpointing an app.

BIBLIOGRAPHY

[1] D. Gilbert, "Microsoft wants to replace your PC with your smartphone ," www.ibtimes.co.uk.

[2] M. Barlett, "Android Random Restart," http://phonetipz.com.

[3] J. Ansel *et al.*, "DMTCP: Transparent checkpointing for cluster computations and the desktop." in *IPDPS*, 2009.

[4] P. Sawers, "Android users app usage ," thenextweb.com.

[5] D. Nguyen *et al.*, "Reducing smartphone application delay through read/write isolation." in *Mobysis*, 2015.

[6] X. Chen *et al.*, "Smartphone background activities in the wild: Origin, energy drain, and optimization." in *Mobicom*, 2015.

[7] K. Kim *et al.*, "Improving energy efficiency of wi-fi sensing on smartphones." in *INFOCOM*, 2011.

[8] W. Hu *et al.*, "Energy-aware video streaing on smartphones." in *INFOCOM*, 2015.

[9] T. Yan *et al.*, "Fast app launching for mobile devices using predictive user context." in *Mobisys*, 2012.

[10] X. Ma *et al.*, "eDoctor: Automatically diagonosing abnormal battery drain issues on smartphones." in *NSDI*, 2013.

[11] J. Ruscio *et al.*, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems." in *IPCPS*, 2007.

[12] O. Laadan *et al.*, "Transparent checkpoint-restart of multiple processes on commodity operating systems." in *ATC*, 2007.

[13] G. Bronevetsky *et al.*, "Application-level checkpointing for shared memory programs." in *SIGOPS*, 2004.

[14] A. Hof *et al.*, "Flux: multi-surface computing in android." in *Eurosys*, 2015.

[15] 0xlab , "Android DMTCP. ," http://github.com/0xlab.

[16] Samsung, "Galaxy Nexus ," www.gsmarena.com.

[17] Monsoon Solutions Inc., "Monsson Power Monitor ," www.msoon.com.